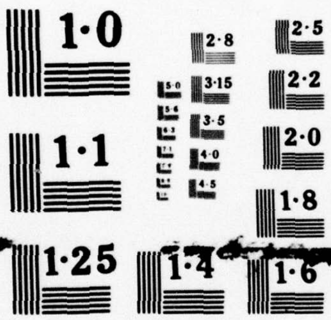
AD-A069 777 ILLINOIS UNIV AT URBANA-CHAMPAIGN COORDINATED SCIENCE LAB F/G 9/2
LEVELS OF REPRESENTATION OF PROGRAMS AND THE ARCHITECTURE OF UN--ETC(U)
AUG 78 B R RAU DAAB07-72-C-0259

UNCLASSIFIED R-819 NL

| OF |
AD
A069777

END
DATE
FILMED
7-79
DDC


1·0
2·8
2·5
3·15
2·2
5·0
5·6
3·5
2·0
4·0
1·1
4·5
1·8
1·25
1·4
1·6

[illegible] AUGUST, 1978 UILU-ENG [illegible]

CSL COORDINATED SCIENCE LABORATORY

LEVELS OF REPRESENTATION OF PROGRAMS AND THE ARCHITECTURE OF UNIVERSAL HOST MACHINES

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) 6 LEVELS OF REPRESENTATION OF PROGRAMS AND THE ARCHITECTURE OF UNIVERSAL HOST MACHINES.		5. TYPE OF REPORT & PERIOD COVERED 9 Technical Report.
		6. ~~PERFORMING ORG. REPORT NUMBER~~ 14 R-819, UILU-ENG-78-2212
7. ~~AUTHOR(s)~~ 10 B. Ramakrishna Rau		8. ~~CONTRACT OR GRANT NUMBER(s)~~ 15 DAAB-07-72-C-0259
9. PERFORMING ORGANIZATION NAME AND ADDRESS Coordinated Science Laboratory University of Illinois at Urbana-Champaign Urbana, Illinois 61801		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Joint Services Electronics Program		12. REPORT DATE 11 August 1978
		13. ~~NUMBER OF PAGES~~ 52
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) 12 56p.		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Universal Host Machines
Emulation
High-Level Language Support

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

The issue of high level language support is treated in a systematic top-down manner. Program representations are categorized into three classes with respect to a host processor: high level representations, directly interpretable representations and directly executable representations. The space of intermediate languages for high level language support is explored and it is shown that whereas the ideal intermediate language from the point of view of execution time is directly executable one, the best candidate from the viewpoint of memory requirements is a heavily encoded directly interpretable representation. → next page

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

097 700 SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered) xlt

20. ABSTRACT (continued)

The concept of dynamic translation is advanced as a means for achieving both goals simultaneously; the program is present in the memory in a compact static representation, but its working set is maintained in a dynamic representation which minimizes execution time. The architecture and organization of a universal host machine, incorporating this strategy, is outlined and the potential performance gains due to dynamic translation are studied.

UILU-ENG 78-2212

LEVELS OF REPRESENTATION OF PROGRAM AND THE ARCHITECTURE OF UNIVERSAL HOST MACHINES

by

B. Ramakrishna Rau

This work was supported in part by the Joint Services Electronics Program (U.S. Army, U.S. Navy and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Levels of Representation of Programs and the Architecture of Universal Host Machines*

by

B. Ramakrishna Rau
Coordinated Science Laboratory
University of Illinois
Urbana, Illinois 61801

*This work was supported by the Joint Services Electronics Program (U.S. Army, U.S. Navy, and U.S. Air Force) under Contract DAAB-07-72-C-0259.

Levels of Representation of Programs and the Architecture of Universal Host Machines

B. Ramakrishna Rau

Abstract

The issue of high level language support is treated in a systematic top-down manner. Program representations are categorized into three classes with respect to a host processor: high level representations, directly interpretable representations and directly executable representations. The space of intermediate languages for high level language support is explored and it is shown that whereas the ideal intermediate language from the point of view of execution time is a directly executable one, the best candidate from the viewpoint of memory requirements is a heavily encoded directly interpretable representation. The concept of dynamic translation is advanced as a means for achieving both goals simultaneously; the program is present in the memory in a compact static representation, but its working set is maintained in a dynamic representation which minimizes execution time. The architecture and organization of a universal host machine, incorporating this strategy, is outlined and the potential performance gains due to dynamic translation are studied.

1. Introduction

1.1. Microprogramming and Interpretation

Microprogramming was originally conceived by Wilkes as a systematic means of implementing the control structure of a computer [1]. The microprogram, which was embedded in a read-only memory, interpreted the instruction set visible to the programmer. In view of the permanence of the microprogram and its transparency to the user, the interpreted instruction set was, reasonably enough, thought of as representing the architecture of the machine.

With the advent of writeable control store, the situation has changed and, yet, the perspective has remained much the same. Writeable control store is viewed as a means of providing a "soft architecture," i.e., one that can be changed dynamically to match the needs of the moment which might, for instance, entail the support of a high level language. The emphasis still is on the interpreted instruction set. The concept of a "soft architecture" is a seductively appealing one on the face of it. However, when subjected to closer scrutiny its advantages are not quite as obvious. In fact, there is one very hard architecture, which is not the architecture best suited to the high level language but that of the microprogrammable machine. The claim that the insertion of an additional level (the "soft" architecture), which must be compiled into and then interpreted, will improve performance is counter-intuitive. It so happens that the claim is correct as evidenced by experience with the Burroughs B1700 [2,3] and the work of Hoevel [4]. That this should be so, remains unconvincing if explained via the concept of "soft" architectures.

Part of the problem is that although the central issue has changed, the perspective has not. An artificial line is drawn upon which sits the conventional machine language. On one side of this line is the domain of high level languages, compilers, interpreters and main memory. On the other side lie the microprograms, nanoprograms, emulators and a host of other micro- and nano- entities. All this terminology from the Wilkes' concept of microprogramming tends to obfuscate the issue which may be phrased as follows: given a certain (open ended) set of high level languages, what is the nature of the host hardware that is best suited to supporting them and what is the process by which programs, written in these high level languages, are supported?

If the set of high level languages is small and if they are similar, the architecture of the host is apt to be high level and closely matched to the high level languages. If, instead, the languages are greatly dissimilar, the host architecture may comprise the "union" of the architectures associated with each language. Generally, this is an extravagant approach, but it has been used when the number of languages is small. A case in point is the IBM System/360 Model 65 which has the hardware and datapaths needed to interpret the System/360 instruction set plus additional instructions which aid in interpreting the 7090 instruction set. Consequently, the host architecture is normally the "intersection" of the architectures tailored to the individual languages. This ensures the generality and flexibility of the host but also implies a somewhat primitive instruction set. A host of this type is termed a universal host machine. A universal host machine does in fact look very much like a microprogrammable machine, but it is better viewed as a special purpose architecture geared toward the task of interpretation. By so doing, one is liberated from the constraints of horizontality, verticality, etc., and can select an architecture with a more high level goal in mind. Decisions on the nature of the format of the instruction set should appear as the outputs of a design process that keeps the overall goal in mind rather than as initial constraints. In view of the above comments, the rest of this paper emphasizes a top-down approach to this issue and an attempt is made to avoid the jargon of microprogramming.

Given a host architecture and a high level language, one could either interpret the latter directly, compile it into the machine language or compile it into an intermediate language which is then interpreted. Hoevel derives conditions under which the last alternative is superior to the other two [5]. These conditions are generally satisfied for the types of universal host architectures that exist or are under consideration. Hoevel does not, however, consider the nature of this intermediate level. This is treated in some detail by Hoevel and Flynn [6].

1.2. Overview

In this paper, an attempt is made to approach, in a systematic manner, the problem of high level language support. The basic assumptions are that, in the future, most programs will be written in a high level language, and that the number and variety of languages will not be constrained. In such an environment, the effectiveness of an architecture is judged not by the performance achieved with a particular language but by the performance over the spectrum of languages. Accordingly, the focus is on the architecture of universal host machines and the nature of the intermediate level language.

It is worthwhile to emphasize that the universal host machine is not offered as a faster way to interpret a single language such as the System/360. A specialized architecture will obviously perform better. The universal host machine is effective only if the objective is to support a large number of languages with equal facility.

Section 2 discusses the various levels of representation of programs. Three levels are isolated: high level representations, directly interpretable representations and directly executable representations. Section 3 considers the space of intermediate representations into which a high level representation may be compiled. This space is shown to have two dimensions: semantic level and the degree of encoding. It is shown that the ideal intermediate representation varies depending on whether importance is attached more to the speed of interpretation or to the compactness of the representation. Section 4 introduces dynamic translation as a means of achieving the two goals simultaneously. The program is stored statically in the compact representation, but its working set is translated dynamically into the representation suitable for speedy interpretation. On the basis of the "principle of locality" it is possible for just a small fraction of the program to be in the dynamic representation and yet find that the majority of instructions that are executed are in the dynamic representation. This dynamic representation of the working set is maintained in a dynamic translation buffer, the organization and management of which is outlined in Section 5. Section 6 discusses the architecture of a universal host machine incorporating a dynamic translation buffer. Finally, Section 7 attempts to evaluate the effectiveness of the dynamic translation buffer.

2. Levels of Representation of Programs

Central to the discussion of the various levels of representation of programs is the concept of binding. We follow Radin [7] in defining a program to be bound with respect to a given automaton if the automaton is able to execute this program correctly. If the automaton is unable to execute the program due to lack of information regarding the syntax and semantics of the program and data structures, then the program is unbound in those aspects with respect to the automaton. Binding is the process of supplying the missing information by augmenting or modifying the program so that the resulting program representation is bound with respect to the automaton.

2.1. Directly Executable Representations

In the context of high-level language support using a universal host machine, it is generally possible to define three levels of program representation. Assuming that the universal host machine is fully defined (i.e., we consider only "hard" architectures) we may immediately define the lowest level; a directly executable representation, (DER), is one which is bound with respect to the architecture of the universal host machine. This is a well defined level since it is possible to test whether a program representation falls into this category by attempting to execute it on the universal host machine. However, as will be seen subsequently, it is possible to specify more than one DER for a program.

2.2. High-Level Representations

At the other end of the spectrum is the high-level representation (HLR). A precise definition of a HLR is impossible, yet it is a fairly well-formed and intuitively meaningful concept. All programs written in "high-level" languages are HLR's. A HLR is designed to be a natural medium for the expression of algorithms. The HLR is characterized by a fairly complex, often context-sensitive syntax involving hierarchical structures and recursive productions. This facilitates the task of the programmer who has available to him features such as block structure and infix notation, but these same features complicate the task of interpretation. Another important property of the HLR is that the mapping between the names of program objects (variables, labels and procedure names) and their type and value is often dynamic. In a language such as ALGOL, [8], the same name can be declared in different blocks and will be associated with a different value in each case. In APL, [9], the type (integer, real, scalar or vector) of the variable is determined by the last value assigned to it. The structure of most high-level languages implicitly assumes the existence of an associative memory; when the name of a variable is encountered, the name must be associated with the corresponding declaration statement which specifies the mapping to the type and value. Similarly, when an explicit transfer of control is to be performed, the statement corresponding to the given label or procedure name must be located before control can be transferred. In fact, associative memory is rarely available at the hardware level and it must then be simulated by performing time-consuming table searches. Lastly, high-level languages have built-in redundancy and long, inefficient symbolic names so as to enhance the intelligibility of the program. These features are all present to aid the programmer in expressing his algorithm but they make the task of the interpreter far more complex. Consequently, the idea of building hardware which can directly interpret a HLR is generally unattractive.

2.3. Directly Interpretable Representations

The approach that is often taken is to compile the HLR into a form more suitable for interpretation. The objective of this translation step is to produce a representation of the original program which is more closely bound to the hardware upon which the program must eventually be interpreted. In particular, the need for an associative memory is removed by performing the associations once and for all during compilation, the mapping from names to type and value is staticized to the extent permitted by the HLR and the hierarchical form of expressions is unravelled so that the order in which the operands and operators appear is more suited to the interpretive process (e.g., Polish notation). The resulting representation is termed a directly interpretable representation (DIR). It is difficult to draw a sharp distinction between the class of HLR's and the class of DIR's. For our purposes, a DIR differs from an HLR in that it does not require an associative memory, it utilizes a simple, context-insensitive syntax and it does not require a preliminary scan (to set up symbol tables and resolve forward references) before the program can be interpreted. It is from this last property that DIR's derive their name. DIR's may also be viewed as program representations for which it is technologically reasonable to build a hardwired interpreter.

The class of DER's is a special subset of the class of DIR's. The HLR could be compiled directly into the machine language of the available hardware. Since any computation that is to be performed must eventually be performed at this level anyway, it might be expected that this strategy would be optimal from the point of view of execution speed. Assuming, for the time being, that the host computer possesses directly addressable memory of a single speed (i.e., a one-level memory hierarchy) and that the amount of memory is unlimited, the strategy of compiling directly into the machine language is, in fact, optimal.

3. The Space of Intermediate Representations

3.1. One Dimension: The Semantic Level

In practice, the amount of memory is limited and it is desirable that the output of the compiler be compact so as to permit the execution of larger HLR programs. One might exploit the manner in which compilers generate code (by substituting similar sequences of machine code for all occurrences of the same terminal symbol) by replacing every such sequence by a procedure call to a routine that performs the same function. In other words, instead of generating a somewhat specific sequence of code in each instance, the compiler generates a procedure call to a generalized procedure along with the arguments for the call. Since code is not repeated, the program representation is substantially more compact. But a price is paid in execution time. Firstly, there is the added overhead which is the result of using a general procedure and having to pass parameters. Secondly, one must forgo any local optimization that the compiler might have performed upon the individual sequences by recognizing the context in which each occurred. Often, however, compilers are too naive to perform such optimization particularly if the high-level language and machine language are greatly removed in their syntax and semantics. Under such circumstances, the optimization penalty resulting from modularizing the DER is minimized. The resulting procedurally structured DER, (PSDER), though still a DER, is a different level of representation. The sequence of procedure calls is semantically identical to a sequence of instructions in a language with more powerful operations. The PSDER is a level of abstraction that is conceptually removed from the machine language. The binding of this abstract level is performed explicitly via the procedure calls and, consequently, the PSDER is directly executable. Nevertheless, it is semantically at a higher level than is the expanded machine language representation. Note that there are a number of members in the class of DER's - the expanded machine language representation and a number of PSDER's of varying semantic level. Depending on the mismatch between the HLR and the machine language, the PSDER can lead to substantial memory savings without a proportionate degradation in execution speed.

The existence of a two-level memory hierarchy consisting of a small, fast first level memory along with a large and relatively slow second level argues more strongly for the use of a PSDER instead of an expanded machine language representation. The use of a PSDER partitions the program space into two parts - the part consisting of the PSDER procedure calls and the part consisting of the semantic procedures. Generally, each reference to the former part will generate a large number of references to the latter part. The latter part, consisting of the semantic procedures is the ideal candidate for placement in the fast memory level since it will be much smaller than the other part and be more likely to fit into the small memory. Since the majority of references are to this part, they will be accessed at the speed of the fast memory and the average access time will be reduced.

In contrast, if the expanded machine language representation is employed, the program will be much larger and it will not be possible to isolate a portion of the program that is small enough to fit in the first level of memory and which also corresponds to a major portion of all memory references. A strategy in which the fast memory level is used as a transparent cache on the slow level would probably be more effective. Even with such an organization, the high density of references to the semantic procedures would make the PSDER more attractive. The existence of a two-level hierarchy, therefore, makes it beneficial, both in execution time as well as in space, to represent the program as a PSDER. This issue is treated in detail by Hoevel [5] who derives conditions under which the use of an intermediate language is better, in time and space, than compiling directly into machine code or interpreting the high-level language directly.

3.2. Another Dimension: The Degree of Encoding

A further reduction in memory requirements can be achieved by compiling down into a DIR rather than into a DER. A DIR may be viewed as having evolved from a PSDER in the manner described below. In general, an instruction at any level of representation must provide three items of information: an algorithm for specifying and accessing each of the operands, an algorithm that specifies the operation upon these operands and an algorithm that specifies which instruction to execute next. Corresponding to a statement or part of a statement in a HLR would be a sequence of procedure calls to procedures which compute the memory addresses and fetch the operands to internal registers or stacks, a call to a functional procedure which operates upon the operands, possibly one or more calls to store results and, finally, perhaps a call to a procedure which would compute the address of the next PSDER instruction. In certain instances, the action of the operand accessing procedure or that of the functional procedure might be simple enough not to warrant a procedure call. Instead, a machine language instruction could be placed in-line with the other calls. Assuming a sufficiently wide variety of these procedures, it is possible to reflect the semantics of a HLR by creating arbitrary "instructions" at the PSDER level by stringing together the required calls (along with their arguments) in open-ended combinations.

Alternatively, one could select a relatively small set of permutations of procedure calls which are deemed to be sufficient for the application at hand. The representation could be made more compact by lumping together all the procedure calls into one field followed by the arguments for all the calls. This newly created field would be a surrogate for the sequence of calls and would serve to specify which procedures should be executed and in what order. This field would be the equivalent of the opcode field in conventional machine instructions and what used to be the arguments of the calls are now equivalent to the operand fields of the conventional machine instruction. The size of the opcode field is determined by the number of distinct permutations of calls that were selected. Note that it is the number of permutations (not combinations) that matters since the correspondence between the procedures and arguments must be specified. Alternatively, the opcode field could merely specify the combination of procedure calls and a format field could specify the order in which the procedures are to be invoked. Without overly restricting this number it will generally be the case that the number of bits needed to specify the opcode field is much less than the number of bits needed for the corresponding machine language procedure call instructions which consist of a machine language opcode (specifying a procedure call) and a pointer to the procedure. Consequently, this representation is more compact than a PSDER.

However, the representation is no longer a DER. An instruction in this new representation does not belong to the machine language and cannot be directly executed. Instead it is interpreted by an interpreter (expressed in a DER) which must fetch each instruction, isolate the opcode field, use this to determine the sequence of procedures which must be executed and activate them in the correct order. The consequent increase in the execution time is the price paid for the memory savings achieved by using a DIR. There is an additional and less obvious execution time penalty in using a DIR rather than a PSDER; by restricting the set of permutations of procedure calls which are coalesced into a permissible DIR instruction format, one reduces the flexibility of the representation and the ability to reflect the semantics of the HLR. This in turn forces the compiler to generate spurious instructions and movement of data. Elson and Rake [10] provide an example of the inefficiencies that can result when compiling from PL/1 to 360 machine language, as a consequence of the "distance" between the semantics of the two languages. Table 1 gives an example of a sequence of PSDER procedure calls which are combined to form a PDP-11 type of instruction and further compressed into a System/360 RX type of format.

The level of a PSDER can be raised by increasing the complexity and variety of the procedures until the PSDER is semantically very close to a HLR. In the case of a DIR one can, analogously, increase the complexity and variety of the opcodes, addressing modes and branch instructions. In addition, one can also increase the number of formats until the DIR approaches the PSDER in semantic flexibility.

A DIR, as we have noted, is an encoding of an equivalent PSDER. The extent of the encoding provides a second dimension in the space of program representations (the first one being the semantic level). It has been demonstrated that the use of information theoretic coding techniques can reduce dramatically the size of a program representation [11-13,3]. Wilner states that memory requirements can be reduced by 25 to 75 percent and Hehner claims program compaction by up to 75 percent.

The simplest form of encoding involves the use of fields which are packed together and allowed to span the boundaries of the units of memory access (e.g., words or bytes). Typically the size of each field is fixed and large enough to specify all possible alternatives. Some economy can be achieved by using contextual information when selecting field sizes; for instance, the scope rules of the HLR limit the number of variables that may be referenced from within a given contour [14]. The operand specification field needs only as many bits as are needed to select from amongst these variables. The field length is variable but fixed within any single contour. A more sophisticated encoding of the Huffman type [15] may be employed by measuring the frequency of occurrence of each operator and operand in the static representation of the program. Often occurring items are represented by fields of shorter length thus decreasing the average number of bits needed to represent an item. It is possible to restrict the permitted field lengths to a small number of selected lengths. This simplifies the decoding problem without sacrificing much by way of memory efficiency [3]. The idea of frequency based encoding may be generalized by considering the frequency of occurrence of pairs, triples, etc., rather than single operators and operands [11-13]. Furthermore, contextual information and frequency information may be employed simultaneously to construct a separate frequency based encoding for each contour.

With increasing degrees of encoding, the size of the program representation decreases and significant memory savings can be achieved.

However, the execution time can increase sharply. With packed, but otherwise unencoded, fields, all that the interpreter for a DIR need do is to mask and extract the field. With a contextually encoded representation, the interpreter must keep track of the various field sizes as the contour changes and refer to the current field size before extracting the field. Decoding a frequency based encoding entails traversing a decoding tree guided by an examination of the encoded field. This also increases the amount of memory occupied by the interpreter. An encoding based on the frequency of pairs of fields would require a separate decode tree for each possible predecessor field. The same is true when context and frequency are both used; a separate decode tree is needed for each context. The use of sophisticated encoding strategies will normally require the use of DIR's rather than PSDER's since it is difficult to provide such facilities at the hardware level and yet retain the flexibility needed to adapt to varying frequency statistics. The variable length opcode field in the Burroughs B1700 is an example of frequency based encoding being applied at the machine language level [16].

3.3. Summary

Summing up the arguments put forward, we find that it is advisable to compile the HLR into some lower level representation which is more closely bound to the machine language in its syntax and semantics. In particular, the representation emitted by the compiler should not assume the existence of an associative memory to map names to values (names should be bound to memory addresses in a virtual machine as far as possible), the hierarchical or tree form of the HLR should have been translated to a sequential form and HLR redundancies, such as symbolic names of unbounded length, should have been replaced by numerical tokens or, if possible, memory addresses. The effect of the compilation step is to factor out large amounts of computation, which would otherwise have had to be performed repeatedly each time a statement in the HLR was interpreted, by performing it just once before the interpretation phase. At the lower end, the expanded machine language representation was found to be unsatisfactory when the HLR and machine language are widely different in their syntax and semantic capabilities. (In Section 6.1 we see why this will generally be the case.) In such circumstances, the use of a PSDER or DIR can lead to significant memory compaction which, in a two-level hierarchy, can also result in a reduction in the execution time. This is achieved by factoring out common code - a space-time dual of the effect of the compiler. Available, then, as an intermediate representation for compilation into and subsequent interpretation or execution is a range of representations from the flexibly formatted and highly encoded DIR's at the high end to the functionally modest PSDER's at the low end. For the same functional capability, there exists a trade-off between the compactness of a DIR and the execution speed of the PSDER.

The space of representations may be graphically represented as in Figure 1. The vertical dimension is a measure of the syntactic and semantic complexity of the representation. The horizontal dimension specifies the complexity of the encoding. A point in the space denotes a representation. In general, the size of a program decreases with increasing distance of the representation from the origin, but the size of the interpreter and semantic routines increases although by a smaller extent. Assuming a two-level memory hierarchy, the interpretation time may be expected to decrease in the vertical direction with increasing level. At the same time, the compile time will decrease since it, presumably, is easier to compile into a higher level intermediate level than it is to compile into one which is greatly removed from the HLR. As one moves to the right, both interpretation and compilation time may be expected to increase. The increase in compilation time is caused by having to compile first into an unencoded form followed by an encoding step.

If one is concerned only with the size of the intermediate representation and the interpretation time, the former consideration would indicate the use of a highly encoded DIR of a level as high as can be tolerated from the point of view of interpreter size. The latter consideration would indicate the use of a PSDER, once again of as high a level as the size of the semantic routines will permit. The size of the semantic routines and interpreter is important since they must fit into the faster, smaller level if high speed interpretation is to be achieved. In the next section we shall present a method of simultaneously fulfilling these contradictory requirements of high speed interpretation and a compact intermediate representation of the program.

4. Dynamic Translation of Program Representations

A characterizing property of a compiler is that whatever binding it does persists over the entire period of execution of the program. The interpreter must complete whatever binding remains. However, this binding persists only over the period of execution of one instruction and must be repeated each time that instruction is executed. From the point of view of persistence of binding, the compiler and interpreter are at opposite extremes. We introduce the notion of a dynamic translator, the persistence of whose binding lies in between that of the compiler and the interpreter. In other words, once the dynamic translator binds an instruction (totally or partially), it remains bound over a period of time that spans a certain number of successive executions of the instruction. Such a strategy assumes, of course, that the program is not self-modifying - an assumption that is generally valid when programs have been written in high-level languages.

One can conceive of a hierarchy of representations each with a different level of binding and degree of persistence: the source program which exists until destroyed, the DIR which lasts until the source is modified, the link-edited version which exists for one execution of the program, possibly a number of lower levels, each increasingly bound and persisting for decreasing fractions of the program execution period and, finally, a completely bound representation which lasts for the duration of just one instruction execution. In addition to the fact that the more tightly bound representation exists for less time, it should be noted that smaller fractions of the program will be, at any one time, in the more tightly bound forms. Thus the source program always exists, but only the procedures which are to be used during this run might be in link-edited form and only the instruction which currently is being executed is totally bound.

The significance of the dynamic translator is that it creates the possibility of simultaneously achieving high speed interpretation and a compact static intermediate representation. Since the binding performed by a dynamic translator persists over a number of executions of an instruction, the time spent in binding is spread out over those instructions, thereby reducing the average time spent in binding per instruction executed. It is possible then to use a highly encoded DIR without increasing the interpretation time by very much if the binding is made to persist over a sufficient number of successive executions of the same instruction. This persistence of binding is effected by saving the bound representation of the instruction which will be less compact than the encoded DIR version. Attempting to retain this bound version for extended periods of time for a number of different instructions will entail the use of large amounts of memory. In fact, if the bound version were never discarded, one would soon obtain and have to provide storage for a translated version of the entire program, thereby defeating the purpose of using an encoded DIR.

The effectiveness of the dynamic translator hinges on the ability to save the bound representation for just a short period of time which, nevertheless, spans a large number of executions of the instruction. The existence of loops and recursive calls would seem to make this possible. In fact, the more general "principle of locality" states that over any interval of time, the vast majority of memory references are concentrated on a small subset of the address space. This principle has been empirically validated over and again [17-19] and is the fundamental justification for the existence of cache memories [19-21] and virtual memories [22,23]. The fraction of the address space that is currently being referenced heavily is termed the working set [18]. The function of the dynamic translator is to maintain in the dynamic translation buffer (DTB) a representation of the instruction working set that is more tightly bound than the static representation. If the size of the DTB is sufficiently large and if the contents of the DTB are selected carefully, it will be found that a large fraction of all instructions executed will be present in the DTB. This fraction is termed the hit ratio. When the hit ratio is close to unity, most instructions when executed will be found in the more tightly bound representation. The time penalty associated with binding will be experienced only rarely and will not be a major factor in determining the execution time. If, at the same time, the size of the DTB is small in comparison to the size of the loosely bound representation, the memory requirements will not have been increased substantially and the conflicting requirements of a compact representation and low execution time will be met simultaneously.

The concept of a DTB is close to that of the dynamic address translation mechanism provided with virtual memories. When addressing a virtual memory, the virtual address must be bound to a physical address. This involves indirection through one or more segment and page tables on each memory reference. This overhead is reduced by retaining in an associative array the mapping between the virtual and physical addresses for the pages which have been referenced most recently. The DTB may be viewed as a cache on a virtual memory in which the program is stored in the more tightly bound representation.

When the dissimilarities between the representations corresponding to minimum execution time and minimum storage requirements, respectively, are great, it is possible that a number of levels of dynamic translation will be required. However, in the rest of this paper, we shall concern ourselves with only one level of dynamic translation. Typically, three different representations are of interest: the HLR in which the program is written, the _static_ (intermediate) _representation_ into which it is compiled and the _dynamic representation_ which is obtained by dynamically translating the static representation of the working set. Of these, only the latter two will be in the directly addressable memory during execution.

The use of dynamic translation permits the decoupling of the design decisions involved in selecting the intermediate representation. The static representation may be selected solely to minimize the size of the program. Ideally, it should be a high level, highly encoded DIR. The dynamic representation, on the other hand, should be selected to speed up execution and should, ideally, be a high level PSDER. However, the organization and management of the DTB can place constraints upon the static and dynamic representations. These issues are discussed in the next section.

5. Organization of the Dynamic Translation Buffer

5.1. Memory Management

One factor that strongly influences the choice of the static/dynamic pair of representations is the size of the unit of allocation of space in the DTB. With an arbitrary choice of the static/dynamic pair, one static instruction could require an arbitrary amount of space in the DTB to store the corresponding dynamic version. This could arise if, for instance, the static representation permits a variety of formats with a widely varying number of operands. In such a case, the amount of space that would have to be allocated in the DTB could be different for each static instruction and the DTB would have to operate under a variable allocation policy. One problem with variable allocation policies is that the replacement policy is complicated since the choice of what is to be replaced in the DTB is influenced by the amount of space needed by the incoming information. Furthermore, the memory fragmentation [24] that results will require time-consuming garbage collection with most replacement policies. Such overheads could be intolerable at the level under consideration.

The use of a fixed allocation policy simplifies the replacement policy considerably but at the cost of constraining the static and dynamic representations. The amount of variability of the instruction formats of the static representation must be limited to ensure that the dynamic translation of a static instruction will fit into the unit of allocation. The dynamic representation in turn should be well matched semantically to the static representation to ensure that a static instruction when translated into its dynamic version will not need more space than the unit of allocation.

Assuming that the static representation is a DIR and that the dynamic one is a PSDER, they must be such that for each opcode or address mode in the DIR, a corresponding semantic routine exists in the PSDER so that a one-to-one correspondence exists between fields in the DIR instruction and procedure calls in the dynamic version.

An alternative is to permit a variable allocation with fixed size increments. When a translated DIR instruction requires more than the unit of allocation in the DTB, another memory block is allocated in a secondary overflow area and is linked to the primary unit of allocation. If the frequency of occurrence of overflow is low, i.e., the primary unit of allocation is chosen judiciously, this scheme does not incur significant overhead and yet permits more flexibility in the choice of the static and dynamic representations. Garbage collection, too, is greatly simplified.

5.2. Organization

The organization of the DTB is, for the most part, similar to that of a conventional cache [19,21,25]. It consists of four memory arrays (Figure 2). The first two, the associative tag array and the address array, are jointly known as the associative address array, one half of which contains the address of the DIR instruction (the associative tag), while the other half contains the address at which the PSDER translation is to be found. The third array, the buffer array, contains the PSDER instructions. This array will normally occupy a predefined portion of the machine's directly addressable memory. Ideally, the DTB should be fully associative, i.e., a DIR-PSDER address pair may be placed anywhere in the associative address array. This permits the replacement policy complete flexibility in deciding what should be replaced and results in a higher hit ratio. However, full associativity implies either that costly hardware be used to search the entire address array in parallel or that relatively slow sequential searches be performed. The compromise employed in most cache designs is to use set associativity [25], generally of degree 4. The DIR instruction address is hashed to select a unique set of four address array locations. These four are searched in parallel using the DIR address as the associative tag. If the required DIR-PSDER address mapping is not present, one of these four locations must be used to store the mapping. The one selected for replacement is that which was used least recently. The replacement array keeps track of the ordering of each set by recency of use. Set associativity of degree 4 has been found to be nearly as effective as full associativity [19].

In cache organizations the pointer into the buffer array is implicit, i.e., the address at which the match is found in the associative address array is used to calculate the required buffer array address. Thus the second array (containing PSDER addresses) is not physically present. The presence of this array, and as part of the processor's directly addressable memory, makes it possible to change the unit of allocation in the buffer to accommodate the needs of different HLR's. Variable allocation policies, too, are supported by this feature. The access time to the PSDER instructions might be increased (depending on the implementation) since two arrays must be accessed before the buffer address is obtained.

6. Architecture of the Universal Host Machine

The desirable architectural features of a universal host machine, (UHM), fall into two broad categories: those features that are generally useful in the task of interpretation, and those that are specific to a UHM that incorporates a DTB. The former category has been discussed at length elsewhere [26]. We shall content ourselves in this paper with merely classifying these features into broad categories without dwelling on the implementational alternatives. Instead, we shall concentrate on the architectural implications of using a DTB.

6.1. General Features

Any language, be it a HLR or DIR, makes certain assumptions about the virtual machine to which it is bound:

1. the ability to parse or interpret the syntax of the language,
2. the nature of the memory space, i.e.,
   a. the number of memory spaces, e.g., registers, control store and main memory,
   b. the type of memory - associative in the case of HLR's and directly addressable for DIR's,
3. the legal data structures, with respect to
   a. resolution - the smallest item of information,
   b. size - the relationship of other data structures to the unit of information,
   c. structure - the aggregation of simpler data types to form more complex ones,

4. semantic capability, i.e.,

   a. the permitted transformations upon the data structures,

   b. the facilities for specifying named objects, e.g., subscripted variables, record fields in PL/1 and base plus displacement addressing in conventional machine level languages such as System 360,

   c. procedural control structures such as subroutines, coroutines, IF-THEN-ELSE, DO WHILE, etc., constructs.

To cope with these assumptions, a UHM must have the following properties:

1. powerful shift and mask instructions which facilitate the extraction and examination of arbitrary bit strings,

2. instructions that aid in the table look-up that is needed to simulate an associative memory,

3. i) high memory resolution, i.e., the ability to view the memory space as a bit string,

   ii) residual specification of data structures to enable memory to be simultaneously viewed in a more structured fashion,

4. i) good functional resolution, i.e., primitive operations from which functions of arbitrary complexity may be synthesized,

   ii) high parallelism so that performance may be preserved despite the existence of a primitive functional capability,

   iii) structural resolution, viz., the ability to manipulate and reconfigure the data paths and interconnectivity of the functional units at a detailed level,

   iv) residual control over those aspects of the datapath structure which are relatively static.

The functional operations provided in the universal host machine should include those that can be thought of as the "greatest common divisors" of the semantic capabilities that are encountered in all DIR's that the UHM may be called upon to interpret. Considering the diversity of existing and conceivable HLR's, the commonality of the corresponding DIR's will exist only at a rather low semantic level. Performance of the UHM may be retrieved by the provision of a number of primitive functional units which may function concurrently. A shortcoming of most UHM's is that data, in the course of a register-to-register transfer, undergoes just a couple of elementary transformations (e.g., an add and a shift). The availability of a large number of busses and functional units and a powerful restructuring capability would permit the hardware to be configured, on a static or dynamic basis, to reflect the data flow graph of complex operators. As a result, more significant transformations could be performed in one register-to-register transaction. Thus, whereas the compiler binds the HLR down towards the hardware, the ability to restructure the data flow topology binds the hardware up towards the DIR.

Primitive operations, a certain amount of parallelism and a limited restructuring capability are found in horizontally microprogrammable machines. Residual control over these abilities allows for shorter instructions without much sacrifice of power. Although elementary operations are necessary for the synthesis of arbitrary functions, this does not preclude the presence of very powerful features aimed specifically at the task of interpretation. Two examples have been noted above - powerful shift, mask and extract instructions and instructions which support table look-up. To be discussed next are architectural features built around the presence of a dynamic translation buffer. We note that most of the features listed above as desirable are present to a greater or lesser extent in many recent microprogrammable processors [2,27-29]. Consequently, any one of these could, presumably, be used as the basic architecture which is to be enhanced by the addition of a DTB.

6.2. Features Specific to the Use of a DTB

The organization of a universal host machine incorporating a dynamic translation buffer is shown at the block level in Figure 3. If the portion within the broken lines is ignored, we have a conventional host machine possessing two levels of memory. The Instruction Unit 1 (IU1) executes programs expressed in a directly executable representation and specifies the contents of the control word which controls the configuration and activity of the UHM's datapaths and functional units. Typically, the interpreter would reside in the level 1 memory as a DER program. The instruction fetch unit (IFU) determines which DER instruction to fetch next and presents it to IU1 which thereupon generates the control word which comprises the control signals. The instruction set recognized by IU1 and the architecture of this portion of the UHM should reflect the desirable features listed above.

The IU1 corresponds to those instructions that are needed for the general task of interpretation. IU2, on the other hand, recognizes those instructions which are specific to the use of the DTB. The function of the PSDER version of a DIR instruction is to steer control to the appropriate semantic routines and to pass parameters. Consequently, the instruction set recognized by IU2 includes CALL, PUSH and POP instructions. The CALL instruction benefits greatly from the presence of a return address stack.

The PUSH and POP instructions which presuppose an operand stack are useful in passing parameters. The limited capacity of the DTB constrains the dynamic version of a DIR instruction to be as short as possible. Accordingly, the instruction set for IU2 must be of a short, vertical format. In contrast, since the instructions recognized by IU1 must exercise detailed control over the configuration of the data paths, they could be of a long, horizontal format.

The short format instructions come in different flavors to permit the operand specification to be immediate, direct or indirect. Thus, in interpreting a descriptor based DIR, the PUSH instruction would specify the address of the descriptor and the indirect mode, resulting in the operand being placed on the stack for the semantic routine. For a DIR in which the operand is specified by a base and displacement method, the contents of the base register would be placed on the stack by a PUSH using the direct mode and the displacement would be stacked by a PUSH using the immediate mode. The address calculation routine would add the two and fetch the operand to the stack for the semantic routine to use.

The most important short format instruction is the INTERP instruction which exercises the DTB. The operand of this instruction is the address of a DIR instruction in the DIR address space. The INTERP instruction causes this address to be presented to the associative address array of the DTB. If it is a hit, the PSDER translation of the DIR instruction is present in the DTB and control is transferred to that sequence of PSDER instructions. The last instruction in this sequence is another INTERP instruction which transfers control to the PSDER version of the next DIR instruction which is to be executed. When the next DIR instruction is known unconditionally (i.e., the sequential successor or the target of an unconditional branch) the operand of the INTERP instruction is supplied immediately. When the next DIR instruction address has to be computed, the result may be left on the operand stack for use by the INTERP instruction. The INTERP instruction, therefore, must come in two flavors depending on whether the operand is specified immediately or left on the stack. Overall, the short format instruction set is similar to a simple and conventional stack-oriented instruction set except, of course, for the INTERP instruction.

If the hit ratio in the DTB were unity, as it will be while the DIR program is in a tight loop, the execution of one sequence of PSDER instructions would lead directly to the execution of the next sequence. The UHM would then be spending all its time in performing computation related to the semantics of the DIR program instead of performing overhead tasks such as parsing, informatic theoretic decoding and binding which constitute computation that is not inherent in the algorithm of the DIR program but is the result of the mismatch between the representation of the program and the hardware.

Averaged over the entire execution of a program, the hit ratio will, of course, be less than unity. The sequence of actions that result when a miss occurs is as follows (Figure 4): the INTERP instruction presents to the DTB a DIR address for which a match is not found in the associative address array. This causes a trap to the dynamic translation routine, the pointer to which is maintained in a dedicated register, DTRPOINT. Simultaneously, the replacement logic of the DTB chooses the location into which the PSDER translation is to be placed, stores the DIR address in the associative tag array and makes available to the dynamic translation routine the pointer to the location in the DTB at which the PSDER translation is to be stored. The dynamic translator fetches the DIR instruction, decodes and parses it, generates the PSDER translation which it then stores in the DTB at the selected location. Lastly, it sets the ball rolling by transferring control to the first instruction in the PSDER translation. The dynamic translator does slightly more than a conventional interpreter in that it must generate the PSDER translation and store it in the DTB instead of merely transferring control to the semantic routines. In this respect, it has something in common with a compiler. However, since the mapping from DIR to PSDER is almost one-to-one, the added complexity is not significant and is easily masked by the number of times that the task of decoding and parsing is avoided.

The control word is specified by one or the other of the instruction units depending on which one currently possesses control. When IU2 processes a CALL instruction to a semantic routine (which is expressed in long format instructions), control is handed over to IU1. The last instruction in the semantic routine causes a return to the dynamic translation of the DIR instruction and automatically returns control to IU2. IU2 only executes instructions fetched from the DTB. The IFU decides which instruction unit is to be active depending on whether the instruction it is fetching is from the DTB or elsewhere. Since the instructions executed by IU2 are relatively small in number (CALL, PUSH, POP and INTERP), the size of the opcode field in the short format instructions is minimized. Thus, by providing two instruction units, it is possible to have both the small, powerful short format instruction set needed to utilize the DTB effectively as well as the larger and more general instruction set needed to perform the semantics of the DIR instruction.

The DTB is shown in Figure 3 as a separate resource, but the address array and the buffer array would, in fact, form part of either the level-1 or level-2 memories. The former alternative is preferable since the access time to the PSDER instructions would be low whereas with the latter alternative, only the decoding and parsing penalty would be saved. However, the increased access time could be partially compensated for by prefetching the entire unit of allocation in the DTB. The advantage of the latter alternative, of course, is that the required size of the level-1 memory is decreased. Similarly, although the two instruction units are shown as separate resources, any common portions could be shared since the two are not concurrently active.

6.3. Comparison to Nanoprogramming

The concept and use of a DTB has certain apparent similarities to nanoprogramming since both utilize multiple levels of memory and representation. The Burroughs D machine [27] is closer in spirit to the type of architecture that we have discussed. This machine executes both long format and short format instructions (micro- and nano- instructions). The short format instructions either deposit literals into registers or act as a pointer to a single long format instruction which exercises detailed control over the hardware. Using our terminology, this is an example of a PSDER except that the procedures are degenerate ones, one instruction in length. The motivation behind this type of nanoprogramming is the same as that for a PSDER - program compaction and a reduction in the size of the fast memory level.

In contrast, nanoprogramming as it is used in the Nanodata QM-1 [29] is an example of the use of a DIR. The microinstructions are not directly expandable and must be interpreted by procedures written in nanocode. Once again, the reason for the use of such an architecture is to reduce the amount of memory required to store the interpreter for the DIR of the problem program.

Dynamic translation and nanoprogramming differ in three important respects. Firstly, the DTB holds (a portion of) a representation of the user's program whereas the microstore in a nanoprogrammable machine holds a representation of the interpreter for the DIR of the user's program. Thus, they address distinct, though similar, issues. Secondly, the contents of the DTB are dynamically varying. This is necessary since the size of a user program is, typically, too large to fit entirely into the DTB. An interpreter, on the other hand, is relatively small in size and can be fitted into the microstore of a nanoprogrammable machine. The most significant difference, however, is the manner in which an instruction in the user program is dynamically translated as it is fetched to the DTB. No parallel exists in nanoprogrammable machines.

7. Performance Analysis of the Dynamic Translation Buffer

In this section, expressions will be derived for the average DIR instruction interpretation rate as a function of the parameters that affect the performance to the greatest extent. These parameters are:

Hardware dependent

τ_1 - the level 1 access time

τ_2 - the level 2 access time

τ_D - the access time to a DTB or cache (nominally $2\tau_1$)

Language dependent

d - average decode time per DIR instruction

g - average time to generate and store the PSDER version of a DIR instruction (after decoding has been performed)

x - average time to perform the semantics of a DIR instruction

s_1 - average number of level 1 memory references to access the PSDER version of one DIR instruction

s_2 - average number of level 2 memory references to access one DIR instruction

Program behavior dependent

h_c - the average hit ratio in a cache (of stated capacity) used to buffer DIR instructions

h_D - the average hit ratio in a DTB (of stated capacity).

Three cases are of particular interest; the performances of a conventional UHM and that of a UHM equipped with a DTB provide a measure of the benefit derived from a DTB. However, the comparison is not quite fair since a UHM with a DTB has more resources than a UHM without one. It is necessary to compare the performance of the former with a UHM that has roughly the same resources but uses them differently. Therefore, the case of a UHM with an instruction cache on the level 2 memory will be studied. For the sake of simplicity it will be assumed that the instruction fetch and decode are not overlapped and that no instruction prefetch is active. Overlap between operand fetch and other computation is permitted since it is all lumped into the parameter x and is common to all strategies.

1. Conventional UHM

$$T_1 = s_2\tau_2 + d + x.$$

The average instruction interpretation time is composed of three components: the instruction fetch time, the time to decode it and the time spent in the semantic routines.

2. A UHM equipped with a DTB

$$T_2 = s_1\tau_D + (1 - h_D)s_2\tau_2 + (1 - h_D)(d + g) + x.$$

In this case, the normal instruction fetch time is given by the first term, but, on the occurrence of a miss in the DTB, level 2 memory must be accessed for a DIR instruction (the second term). This DIR must be decoded and translated, which accounts for the third term.

3. A UHM equipped with a cache

$$T_3 = h_c s_2\tau_D + (1 - h_c)s_2\tau_2 + d + x.$$

The first two terms account for the average instruction fetch time. Every DIR instruction interpreted must be decoded. For the same capacity for the cache or DTB, h_c will be closer to unity than will h_D since the DIR representation is more compact.

Two important figures-of-merit for the DTB strategy are given by F_1 and F_2 where

$$F_1 = \frac{T_3 - T_2}{T_2} \times 100$$ i.e., the percentage degradation caused by using the DTB as an instruction cache

and

$$F_2 = \frac{T_1 - T_2}{T_2} \times 100$$ i.e., the percentage degradation caused by not using a DTB.

The evaluation of F_1 and F_2 is hampered by the lack of suitable statistics. A number of the parameters are very dependent upon the type of program, the static and dynamic representations and the architecture of the host machine. The figures of merit would have to be evaluated for each specific case. We shall, however, calculate F_1 and F_2 for representative and plausible values of the parameters.

The unit of time is taken to be the access time of the level 1 memory which is also assumed to be equal to one machine instruction execution time. Therefore, $\tau_1 = 1$. τ_D is assumed to be $2 \times \tau_1 = 2$ and τ_2 is assumed to be $10 \times \tau_1 = 10$. g is set equal to $1.5 \times d$ and S_2 is taken to be 1 and S_1 is chosen to be 3. Thus the dynamic representation of one DIR instruction is assumed to be three times as long on the average as the DIR instruction. A study of the literature on cache memories [19,21,30] indicates that a choice of 0.9 for h_c is reasonable for a cache size of 4096 bytes. The effective DTB size is 4096/3 bytes since $S_1 = 3S_2$. A reasonable value for h_D, then, is 0.8. Substituting these values into the above equations gives

$$F_1 = \frac{0.4 + 0.6d}{8 + 0.4d + x} \times 100$$

and

$$F_2 = \frac{7.4 + 0.6d}{8 + 0.4d + x} \times 100$$

where d, the average number of instructions spent in decoding a DIR instruction, and x, the average time per DIR instruction spent in the semantic routines are yet to be specified.

The parameter d is very dependent upon the extent of encoding and the hardware features of the host machine. The provision of powerful field extraction instructions reduces d. However, the use of frequency based encoding increases the number of levels of decoding needed. For each field, for each level of decoding, at least two instructions are needed; the first one extracts the field (or a portion thereof) and increments the program counter by that amount, causing a CASE STATEMENT type of branch to a list of branch instructions. The selected branch instruction must then be executed, thereby transferring control to either a semantic routine or to another routine which continues decoding at the next level. Thus even with a powerful host architecture, d could easily be equal to 10. For simpler host architectures, d might well be twice as large if not more. The parameter x can vary greatly depending on the nature of the DIR and the architecture of the host.

Tables 2 and 3 list F_1 and F_2 for various values of d and x. The figures demonstrate that the DTB does have the potential to improve performance significantly. The actual values of F_1 and F_2 for any given situation must, of course, be evaluated for the specific values that the parameters assume in that particular case. In general, the figures-of-merit decrease as d decreases or as x increases. Thus the DTB is not particularly effective if the task of decoding is trivial or if the time spent in the semantic routines is much greater than the time that would be spent in decoding. This would be true, for instance, in machines with vector instructions which are heavily used.

8. Conclusion

The architecture and instruction set of a processor is determined by the class of languages that will be executed (interpreted) by it, either directly or following compilation. If this class is restricted, the application of the processor is fairly specific and the instruction set will be at a high level and closely matched to the single or small number of high-level languages that are supported by the processor. The several examples of high-level machine designs fall into this category [31-36].

On the other hand, if the class of languages is large and vague, commonality of semantics will exist only at a very low level and the instruction set of the universal host machine will be primitive. Under such circumstances, the high-level language and the machine language are extremely dissimilar and it is more efficient, both in space and time, to interpose an intermediate level, a directly interpretable level, into which the program is compiled and which is interpreted by an interpreter written in the machine language. An intermediate language is characterized by its position in a two-dimensional space of which one dimension is the semantic level of the language and the other dimension is the degree of encoding.

However, the choice of the intermediate language is complicated by the fact that it is possible to trade-off execution time against the size of the intermediate language program representation. The concept of dynamic translation was introduced to overcome this dilemma. The dynamic translator permits the program to be present in a compact, static representation but maintains the working set in a dynamic representation that lends itself to speedy execution. The dynamic translator dynamically translates instructions from one representation to the other as they enter the working set. Expressions were derived for two figures-of-merit of this scheme and they were evaluated for certain typical values of the relevant parameters, demonstrating the potential performance benefits of this scheme.

The decoding overhead of a universal host machine may be reduced either by providing powerful hardware aids to the decoding process or by the use of a dynamic translation buffer which decreases the number of instructions that need be decoded. The former approach requires the addition of random logic whereas the latter approach relies on the use of memory. This fact is expected to influence the cost-effectiveness of the two schemes. Future research will be aimed at gathering statistics which permit a more quantitative evaluation of the cost-performance of various combinations of intermediate representations and universal host machine architectures, with and without dynamic translation buffers.

Acknowledgments

The author would like to acknowledge Michael Schlansker for many profitable discussions which have lead to a number of improvements in this paper.

References

1. M. V. Wilkes, "The best way to design an automatic calculating machine," Manchester Univ. Comput. Inaugur. Conf., 1951, p. 16.

2. W. T. Wilner, "Design of the B1700," AFIPS Conf. Proc., 1972 FJCC, 41, 489-497, Montvale, NJ, AFIPS Press.

3. W. T. Wilner, "Burroughs B-1700 Memory Utilization," AFIPS Conf. Proc., 1972 FJCC, 41, 579-586, Montvale, NJ, AFIPS Press.

4. L. W. Hoevel, "DELTRAN Principles of Operation: A Directly Executed Language for FORTRAN-II," Tech. Note No. 108, Digital Systems Laboratory, Stanford Univ., Stanford, Calif., March 1977.

5. L. W. Hoevel, "'Ideal' directly executed languages: an analytic argument for emulation," IEEE-TC, 23, 8, 1974, 759-767.

6. L. W. Hoevel and M. J. Flynn, "The Structure of Directly Executed Languages: A New Theory of Interpretive System Support," Digital Systems Lab. Tech. Rep. No. 130, Stanford Univ., March 1977.

7. G. Radin, "A note on the concept of binding," IBM Thomas J. Watson Res. Rep. No. RC 3287, Yorktown Heights, New York, March 1971.

8. P. Naur, (Ed.), "Revised report on the algorithmic language ALGOL 60," CACM 6, Jan. 1963, 1-17.

9. K. E. Iverson, "A Programming Language," Wiley, New York, 1962.

10. M. Elson and S. T. Rake, "Code-generation technique for large-language compilers," IBM Sys. Jour., 9,3, 1970, 166-188.

11. C. C. Foster and R. Gonter, "Conditional Interpretation of Operation Codes," IEEE-TC, Jan. 1971, 108-111.

12. E. C. R. Hehner, "Computer design to minimize memory requirements," Computer, 9,8, Aug. 1976, 65-70.

13. E. C. R. Hehner, "Information Content of Programs and Operation Encoding," JACM, 24,2, Apr. 1977, 290-297.

14. J. B. Johnston, "The Contour Model of Block Structured Processes," Proceedings of the SDSPL (SIGPLAN Notices, Vol. 6) Feb. 1971, 55-82.

15. D. A. Huffman, "A method for the construction of minimum redundancy codes," I.R.E., 40,9, Sept. 1952, 1098-1101.

16. A. B. Salisbury, "Microprogrammable Computer Architectures," Elsevier Computer Science Library, New York, 1976.

17. B. S. Brawn and F. G. Gustavsen, "Program Behavior in a Paging Environment," AFIPS Proceedings, 33, FJCC, 1968, 1019-1032.

18. P. J. Denning, "The working set model for program behavior," CACM, 11,5, May 1968, 323-333.

19. K. R. Kaplan and R. O. Winder, "Cache-based Computer Systems," Computer, 6, 3, March 1973, 30-36.

20. D. H. Gibson, "Considerations in Block-Oriented Systems Design," Proc. SJCC, 1967, pp. 78-80.

21. R. M. Meade, "Design Approaches for Cache Memory Control," Computer Design, 10, January 1971, 87-93.

22. T. Kilburn, D. B. G. Edwards, M. J. Lanigan and F. H. Summer, "One-level Storage Systems," IRE Trans. Elec. Comp., 11,2, 1962, 223-235.

23. P. J. Denning, "Virtual Memory," Computing Surveys, 2,3, 1970, 153-189.

24. B. Randell, "A Note on Storage Fragmentation and Program Segmentation," CACM, 12,7, July 1969, 365-369.

25. C. J. Conti, "Concepts for Buffer Storage, "Computer Group News, 2, March 1969, 9-13.

26. S. M. Fuller, V. R. Lesser, C. G. Bell and C. M. Kaman, "The Effects of Emerging Technology and Emulation Requirements on Microprogramming," IEEE-TC, 25,10, Oct. 1976, 1000-1009.

27. E. W. Reigel, V. Faber and D. A. Fisher, "The interpreter - a micro-programmable building block system," AFIPS Conf. Proc., 1972 SJCC, 40, 705-723, Montvale, NJ, AFIPS Press.

28. H. W. Lawson and B. K. Smith, "Functional Characteristics of a Multi-lingual Processor," IEEE-TC, 20, July 1971, 732-742.

29. Nanodata Corp., "QM-1 Hardware Level User's Manual," Second Edition, August 1974.

30. W. D. Strecker, "Cache Memories for PDP-11 Family Computers," Third Annual Symposium on Computer Architecture, 1976, 155-158.

31. J. P. Anderson, "A computer for direct execution of algorithmic languages" Proc. EJCC, 1961, 184-193.

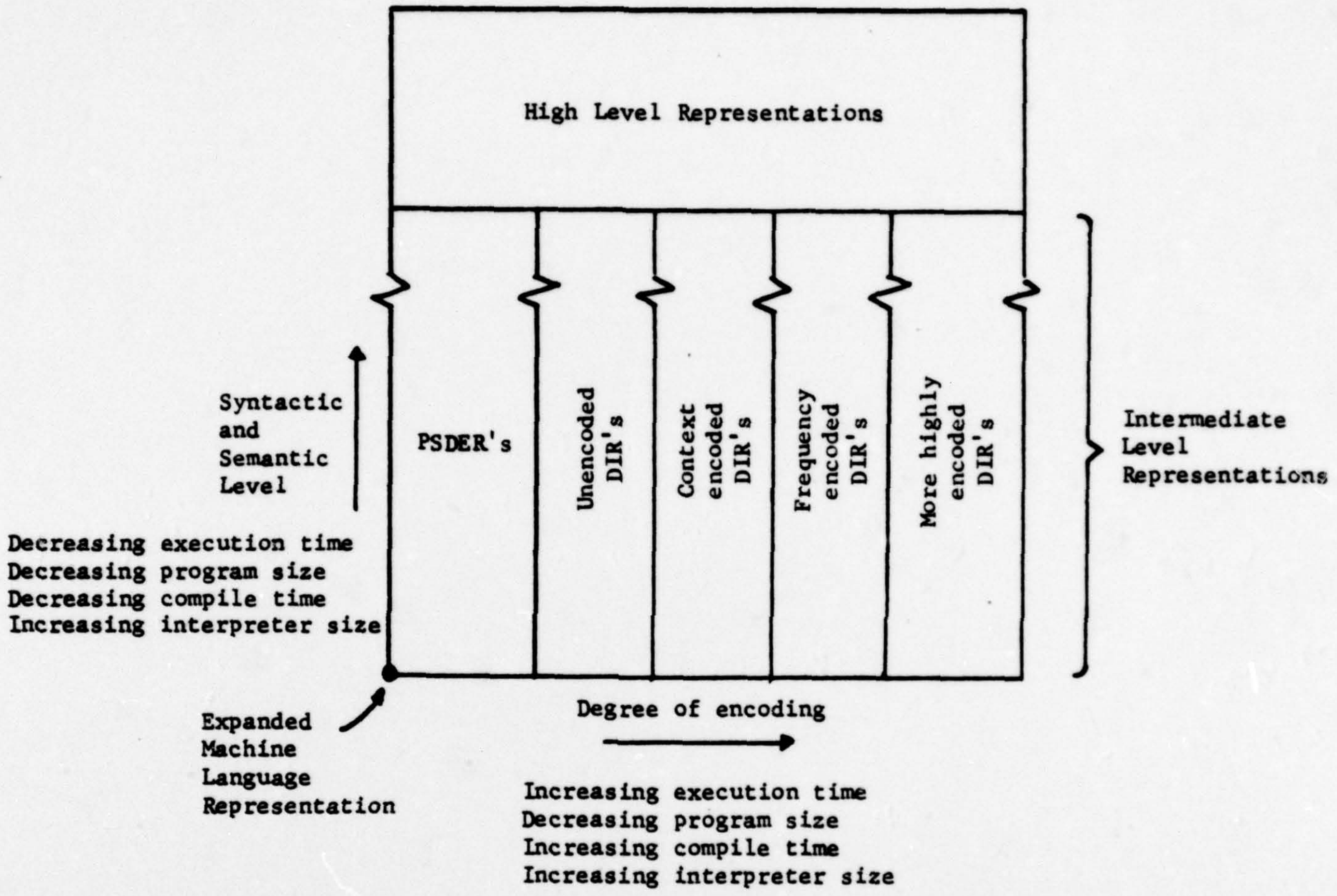
32. Y. Chu, "Introducing the high-level language computer architecture," Tech. Rep. No. TR-227, Comput. Sci. Center, Univ. Maryland, College Park, Maryland, 1973.

33. H. M. Bloom, "Design and simulation of an ALGOL computer," Tech. Rep. No. 70-118, Comput. Sci. Center, Univ. Maryland, College Park, Maryland, 1970.

34. T. R. Bashkow, A. Sasson and A. Kronfeld, "System design of a FORTRAN machine," IEEE-TEC, Aug. 1971, 485-499.

35. M. Sugimoto, "PL/1 reducer and direct processor," Proc. ACM, 1969, 519-538.

36. R. Rice and W. R. Smith, "SYMBOL - A major departure from classic software dominated von Neumann computing systems," Proc. SJCC, 1971, 575-587.



High Level Representations
PSDER's
Unencoded DIR's
Context encoded DIR's
Frequency encoded DIR's
More highly encoded DIR's
Intermediate Level Representations
Syntactic and Semantic Level
Decreasing execution time
Decreasing program size
Decreasing compile time
Increasing interpreter size
Expanded Machine Language Representation
Degree of encoding
Increasing execution time
Decreasing program size
Increasing compile time
Increasing interpreter size

Figure 1. The space of program representations.

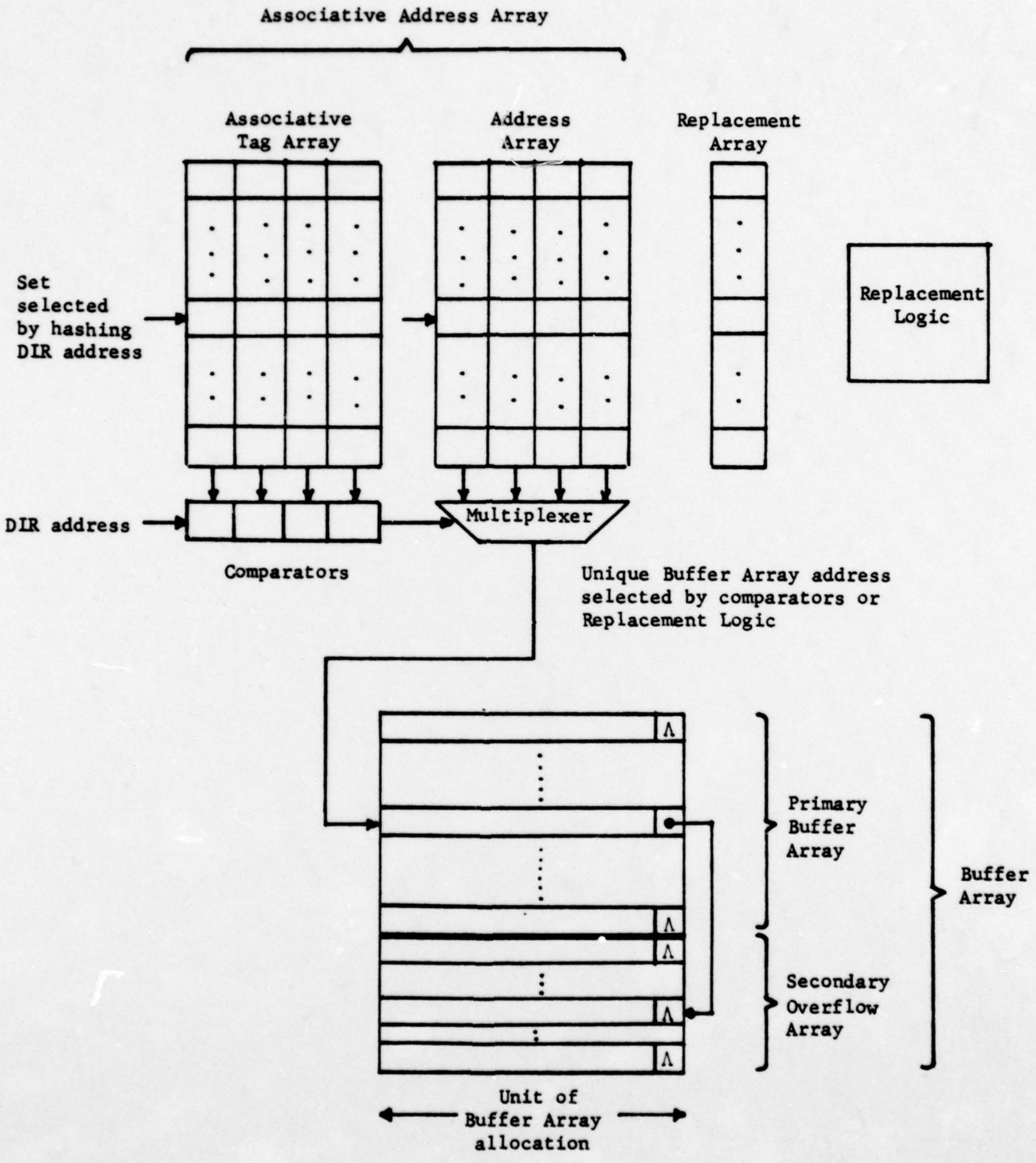

Associative Address Array
Associative Tag Array
Address Array
Replacement Array
Replacement Logic
Set selected by hashing DIR address
DIR address
Multiplexer
Comparators
Unique Buffer Array address selected by comparators or Replacement Logic
Λ
Primary Buffer Array
Secondary Overflow Array
Buffer Array
Unit of Buffer Array allocation

Figure 2. Organization of the Dynamic Translation Buffer.

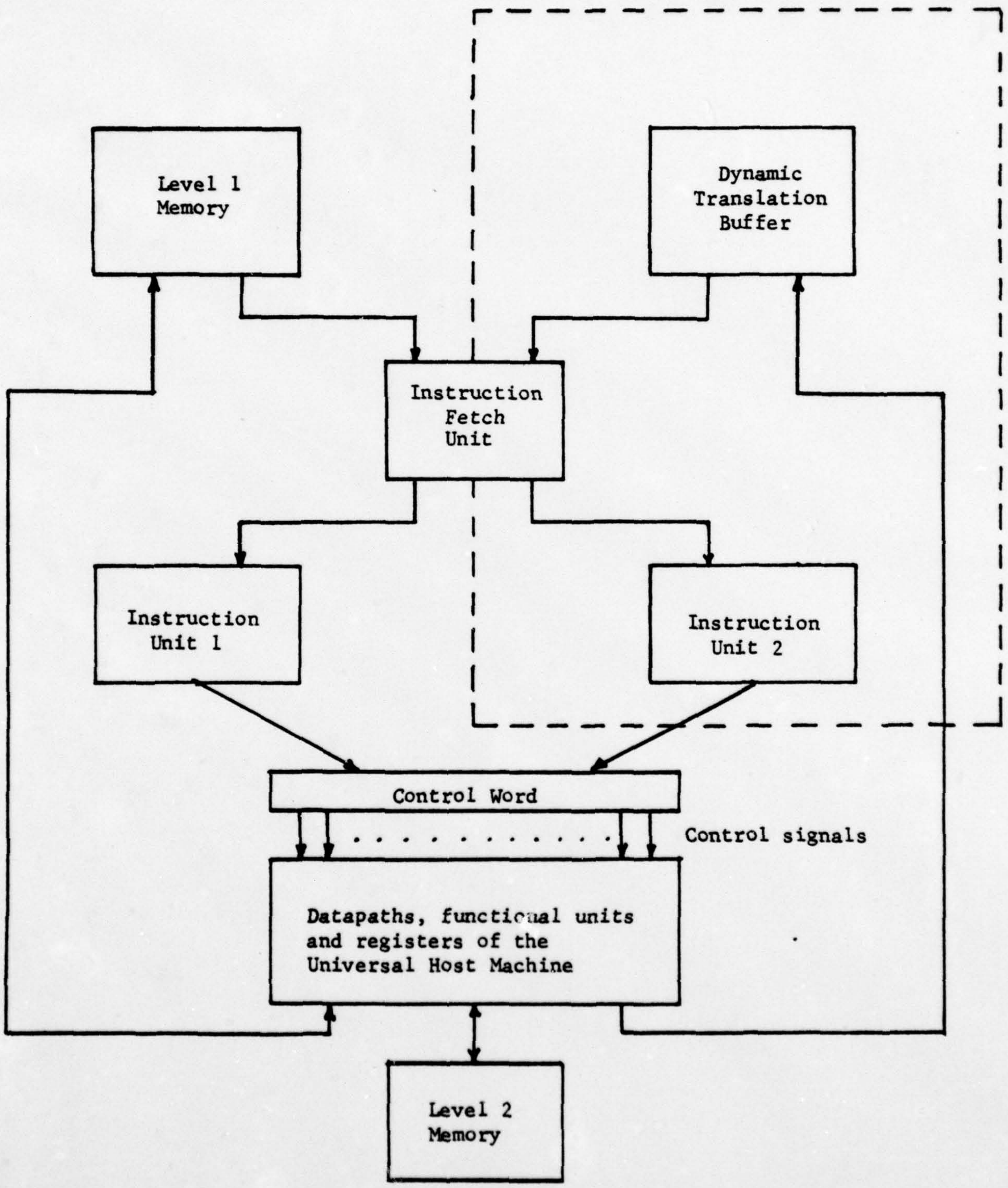
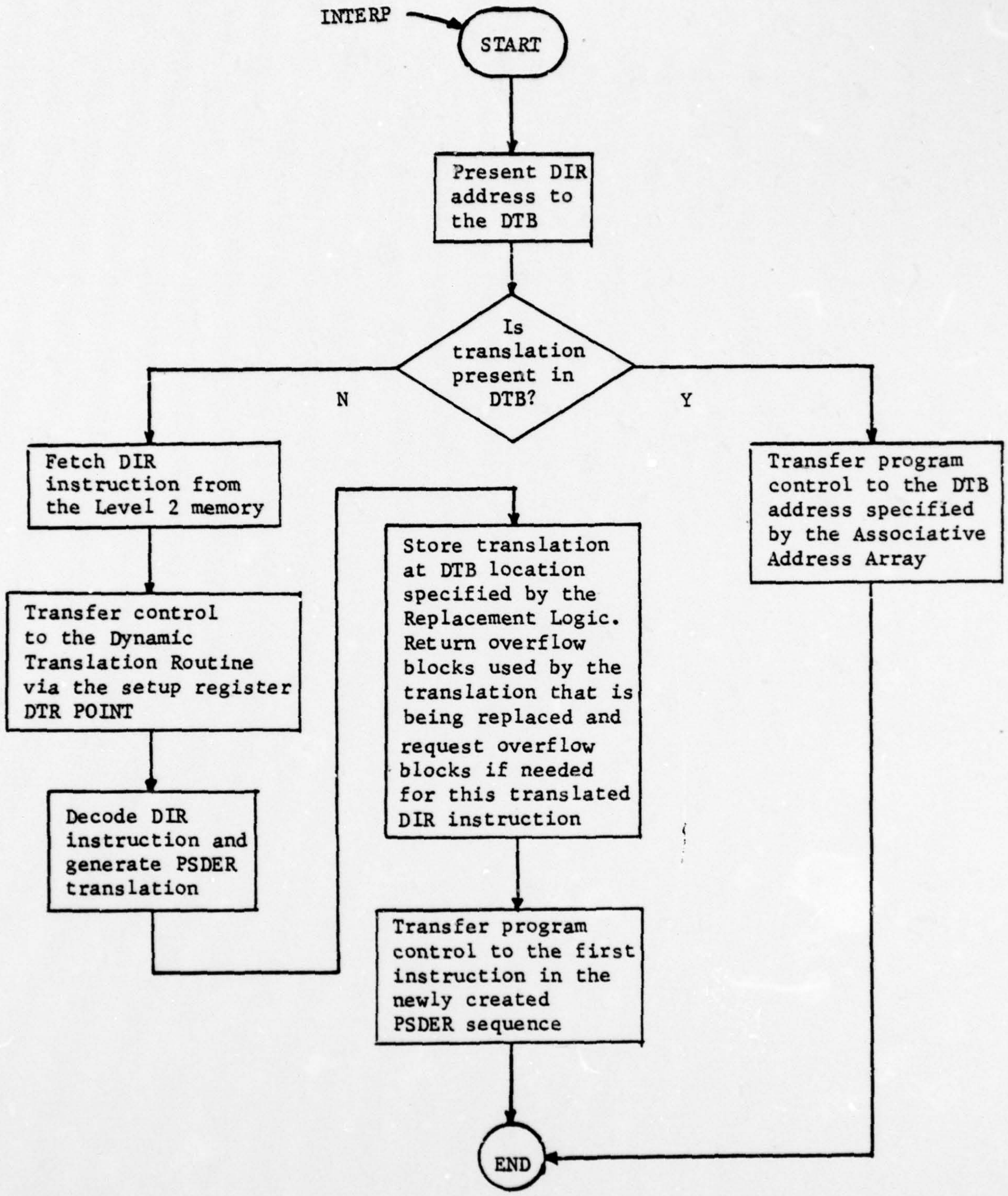
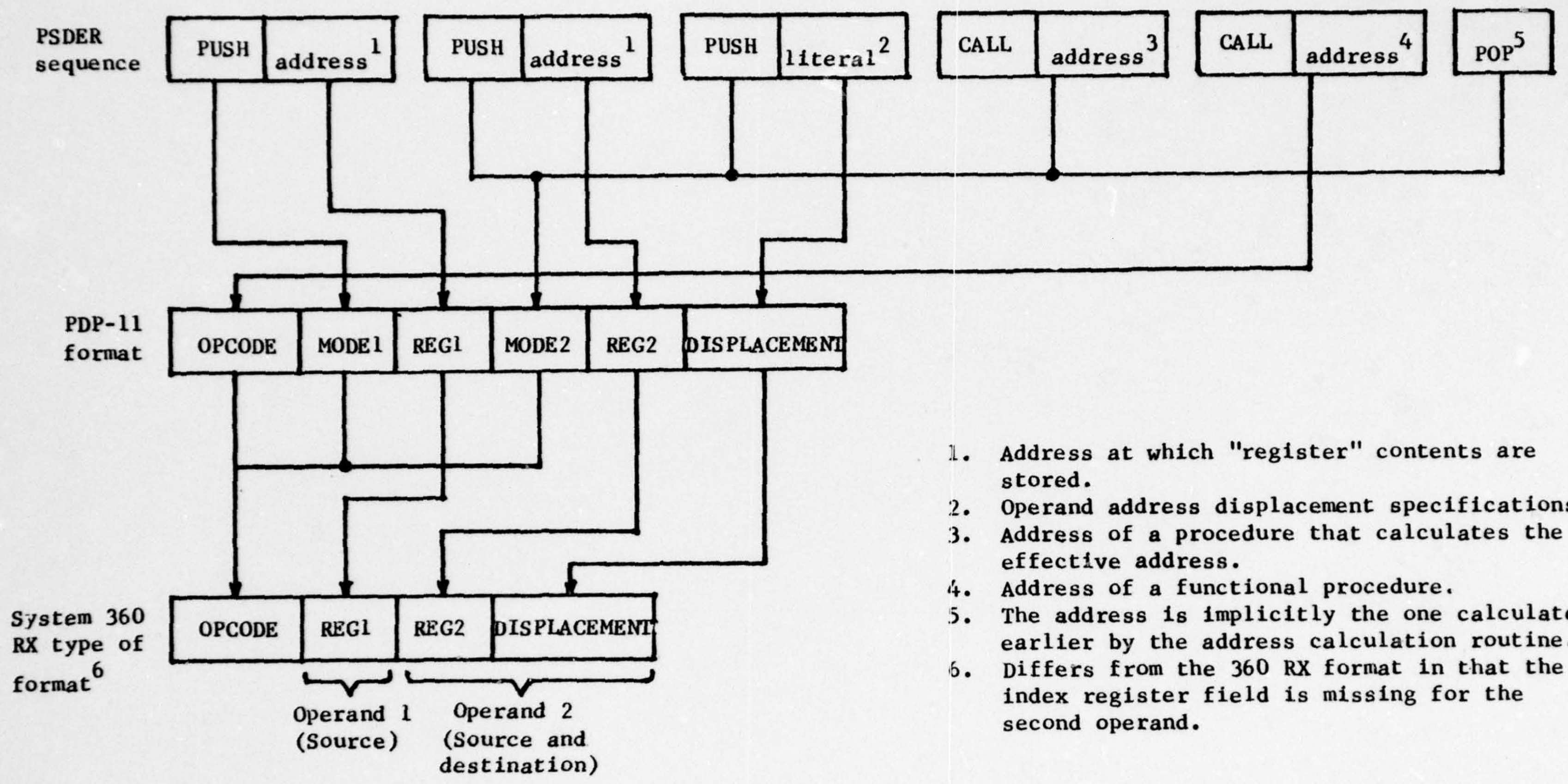


Figure 3. Organization of the Universal Host Machine.



INTERP
START
Present DIR address to the DTB
Is translation present in DTB?
N
Y
Fetch DIR instruction from the Level 2 memory
Transfer control to the Dynamic Translation Routine via the setup register DTR POINT
Decode DIR instruction and generate PSDER translation
Store translation at DTB location specified by the Replacement Logic. Return overflow blocks used by the translation that is being replaced and request overflow blocks if needed for this translated DIR instruction
Transfer program control to the first instruction in the newly created PSDER sequence
Transfer program control to the DTB address specified by the Associative Address Array
END

Figure 4. Flow diagram for the INTERP instruction.



1. Address at which "register" contents are stored.
2. Operand address displacement specifications.
3. Address of a procedure that calculates the effective address.
4. Address of a functional procedure.
5. The address is implicitly the one calculated earlier by the address calculation routine.
6. Differs from the 360 RX format in that the index register field is missing for the second operand.

Table 1. Equivalence of a PSDER sequence to more compact, encoded formats.

d \ x	5	10	15	20	25	30
10	37.65	29.09	23.7	20	17.3	15.24
20	59.05	47.69	40	34.44	30.24	26.96
30	73.6	61.33	52.57	46	40.89	36.8

Table 2. Percentage increase in the average DIR instruction interpretation time due to using the DTB as a cache on the level 2 memory.

d \ x	5	10	15	20	25	30
10	78.82	60.91	49.63	41.88	36.22	31.90
20	92.38	74.62	62.58	53.89	47.32	42.17
30	101.6	84.67	72.57	63.5	56.44	50.8

Table 3. Percentage increase in the average DIR instruction interpretation time due to not using the DTB.